AD-A105 202   KANSAS STATE UNIV MANHATTAN
PROGRAMMING ISSUES IN DISTRIBUTED SYSTEMS,(U)   F/G 9/2
1979   V WALLENTINE
UNCLASSIFIED   DAAG29-78-G-0200
NL

1 OF 1
AD
A105202

END
DATE
FILMED
10 81
DTIC

ADA105202

PROGRAMMING ISSUES IN DISTRIBUTED SYSTEMS*

by

Virg Wallentine
Kansas State University

11 1979

12 7

*To be published in the Proceedings of the Workshop on Network Interprocess Communications, Georgia Institute of Technology, Atlanta, Georgia, November 1980.

This research was supported in part by grant number DAAG 29-78-G-0200 from the Army Research Office and the Army Institute for Research in Management Information and Computer Science.

81 10 2 130

145250

Problem

The programmer in a distributed processing environment must be provided with a set of facilities which permit easy specification of the distributive properties of his/her program. The word program here is used to refer to either the output of a single compilation or the output of independent compilations of program modules which are to be communicating via an IPC. These distributive properties include the specification of the concurrency, data flow, resource requirements (memory, devices, etc.), and intraprogram (intermodule) protocol properties inherent in the execution of a configuration (system) of cooperating software modules. Given a description of these properties, an operating system must be able to distribute the user's program across multiple machines in a manner which is transparent to the programmer. Traditional approaches to providing these facilities include the concurrency support in high-level languages and the resource allocation and concurrency support in conventional operating systems.

Current Approaches

Several high-level languages such as Concurrent Pascal [1] and SP/K [2] have incorporated the monitor [3,4] concept to provide structured concurrency. This concept is excellent in a centralized system but relies on shared data (and therefore shared memory), and is therefore not an appropriate concept on which to base a distributed system. However, an effort is underway at the National Physical Laboratory [5] to distribute a Concurrent Pascal program across loosely coupled microprocessors. The distribution of passive system components (such as monitors) on disjoint machines implies many copy operations for

parameters and also additional active system components (processes) which do not appear in the program text.

A much more appropriate high-level language concept for distributed programs is proposed by C. A. R. Hoare in reference [6]. Each function is a sequential process which is connected to other communicating sequential processes via input/output. This concurrency support is based on data flow and not shared data; therefore, it is not dependent on shared memory. As a result, each function is distributable. However, it seems that buffering of data between processes is necessary to improve performance in distributed systems with slow speed connections. Since the compiler for such a language presumably can generate the resource requirements for the program, since processes are identified by name, and since the protocol between processes is fixed, enough knowledge is available to distribute a set of processes which are compiled together.

A second area of programmer concern for distribution occurs because concurrent program functions (modules) may be separately generated (compiled). These may well be existing programs or just separate functions based on programming style. The interconnection of these modules into a program is dynamic and therefore requires operating system support. In early conventional operating systems, the support for combining these functions into a configuration of communicating concurrent software functions is specified at three levels. First, overlap of CPU and I/O is made available for standard I/O file functions. Second, added concurrency is achieved only with unstructured (low-level) facilities for process creation, naming, and communication. Third, complex job control languages are provided to achieve allocation

of resources to run these functions. In a distributed system, these JCL steps must be synchronized across machines. Complex resource control in a distributed system should certainly not be the programmer's responsibility. This is alleviated by viewing distributed operating systems and their executable programs as cooperating processes. A highly successful system is the Distributed Computing System of Farber [7]. In this system, the structure and distribution of the set of processes is transparent to the user; and a high level of concurrency is achieved without use of low-level process control primitives.

Process naming of cooperating processes is still burdensome to the programmer. The same problem also occurs in current "mailbox" schemes as epitomized by the VAX 11/780 system [8]. The naming or numbering of mailboxes must be known to the programmer or a creating process. This is commonly referred to as the IPC-setup problem, coined by Elliot Organick in reference [9]. The designers of UNIX [10,11] sought to alleviate this problem. They invented the "pipe." In UNIX a user program, running in its own process, may take the place of a file in a manner which is transparent to the original program. Each program may have its standard input and output files replaced by programs, thus building via the UNIX shell arbitrarily long linear chains (a pipeline) of programs. UNIX automatically transfers the data between processes and synchronizes the process as it intercepts the standard input and output file operations.

UNIX "pipes" eliminate the need for process naming and treat concurrency, resource allocation, and interprocess protocol as a data flow problem. Interprocess protocols are treated simply as simplex data streams. The job control language provided by the UNIX shell becomes a

pseudo data flow language and resource allocation is transparent to the programmer. However, there are a considerable number of programmer protocols which are not served by "pipes." As acknowledged in reference [11], "pipes" cannot be used to construct multi-server subsystems.

UNIX will support general interprocess communication protocols, but these are not generated by the shell. These can be programmed as a set of child processes whose "pipes" have been setup by a parent process.

A Research Direction

If we are to be successful in distributing programs across highly distributed systems, we must provide the programmer of dynamically interconnected cooperating processes a job control language (software configuration control) as easy to use as Hoare's communicating sequential processes. It seems that the most promising direction is to extend the concept of the UNIX shell to automatically generate the more complex protocols available to the parent processes previously described. It must then also also be extended to generate (representations of) distributable configurations of communicating processes.

Work in this area is underway at Kansas State University. The project involves development of a Network Adaptable Executive (NADEX) [12]. The attempt is to permit the user to specify data flow at the command level and have the command interpreter generate a distributable software configuration of nodes connected by full duplex data transfer stream connections (DTS connections) to form an undirected graph. In general, a node may be thought of as a process. Each of the connections consists of two independent bi-directional data transfer streams. One

of these streams uses small parameters while the other uses a standard-sized data buffer. The data buffers carry along with them size and status indicators, whereas the parameter buffers contain only a small amount of user-supplied data.

A user program running in a node performs serial buffered READ and WRITE operations in its various connections. The connections are numbered, and the program attaches particular meanings and implements particular protocols for each of its connections. A connection can connect a node either to a user program or to a system process used to access a file or an I/O device. The program cannot tell the difference between these modes of operation. This clearly provides all of the power of the UNIX pipelines while removing the linearity constraint on the structure of the connection graph. Also, the connections are bi-directional so that, for example, a write-request/read-response protocol to access a random file can be implemented.

For these serial buffered READ and WRITE operations, a priori protocol knowledge can be specified to the underlying data flow implementation (buffer control) to enable it to maintain a check for validity of user protocol (in terms of data flow) during execution. This protocol checking is critical in "un-debugged" (user-written) nodes. Examples of such protocol violations occur many times in the facilities of SOLO [13]. Deadlock detection is also performed based on data flow in a configuration which is distributed across machines connected by a network IPC. Multiserver subsystems, such as a data base management system, are implementable as a configuration with multi-connection READ (multiple condition WAITs) and conditional WRITE operations provided on data transfer streams. Interconfiguration

connections are also provided. Finally, the command interpreter and the node interface (PREFIX) provide all the mapping of logical data streams (ports) onto implementation data streams.

REFERENCES

1. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, 1977.

2. Holt, R. C.; Graham, G. S.; Lazowshka, E. D.; and Scott, M. A., "Announcing Concurrent SP/K," Operating System Review, 12, 2 (April 1978).

3. Brinch Hansen, P., Operating Systems Principles, Prentice-Hall, 1973.

4. Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," Communications of the ACM, 17, 10 (October 1974).

5. Dowson, M., "The DEMOS Multiple Processor Technical Summary," National Physical Laboratory Technical Report, NPL Report 101, April 1978, Teddington, Middlesex TWII OLW, UK.

6. Hoare, C. A. R., "Communicating Sequential Processes," Communications of the ACM, 21, 8 (August 1978), pp. 666-677.

7. Farber, D. J., et al., "The Distributed Computing System Digest of Papers," COMPCON 73, February 1973, pp. 31-34.

8. Digital Equipment Corporation, VAX/11 Software Handbook, 1977.

9. Organick, E. I., The MULTICS System: An Examination of Its Structure, MIT Press, 1972.

10. Thompson, K., and Ritchie, D. M., "The UNIX Time-sharing System," Communications of the ACM, 17), 7 (July 1974), pp. 365-375.

11. Ritchie, D. M., "A Retrospective in UNIX Time-sharing System," The Bell System Technical Journal, 57, 6, part 2 (July--August 1978).

12. Young, R., and Wallentine, V., "The NADEX Core Operating System Services," Kansas State University, Dept. of Computer Science Technical Report, No. CS 79-21, November 1979.

13. Brinch Hansen, P., "The SOLO Operating System," Software Practice and Experience, 6, 2 (April--June 1976), pp. 141-206.

ATE

LMED

-8